Resource Constrained Test Case Prioritization with Simulated Annealing in an Industrial Context

Eric Felding
Linköping University
Linköping, Sweden

Per Erik Strandberg
Westermo Network Technologies AB
Västerås, Sweden

Nils-Hassan Quttineh
Linköping University
Linköping, Sweden

Wasif Afzal
Mälardalen University
Västerås, Sweden

ABSTRACT

We need to find an effective prioritization of regression test cases due to their growing number. This may happen on parallel test systems and software branches. We compared regression test prioritization approaches against several goals of importance in an industrial context. We experimentally compared different simulated annealing approaches, hypothetical ideal and worst prioritizations, as well as reference prioritizations such as random, historical failure rate, age, etc. These were evaluated against a heuristic metric that combines several factors, as well as reference metrics such as failure count, days since last execution, etc. By simulating resource starvation in terms of available time, we found that some approaches rapidly degraded, e.g., by only prioritizing recently failed tests, the average number of nights since last execution was about five times as bad as for a random selection. The simulated annealing approach with large search space and many iterations came out best for many metrics. Interestingly, the poorest prioritization was achieved by aiming at diversity, and the coverage-based prioritization was poor at finding failures.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

software testing, test case prioritization.

ACM Reference Format:
Eric Felding, Per Erik Strandberg, Nils-Hassan Quttineh, and Wasif Afzal. 2024. Resource Constrained Test Case Prioritization with Simulated Annealing in an Industrial Context. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605098.3635971

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3635971

1 INTRODUCTION

Software (SW) is everywhere and regularly updated in agile development processes. Regression testing (RT) is to re-test the SW to find out if changes introduced bugs. For embedded systems, the testing should include hardware at some point of the development process, resulting in a test system. On such test systems, test suites can take hours or days to run and with frequent SW changes, and with added test cases, there is not enough time to test everything for each change. Prioritizing test cases thus becomes an interesting problem and with it comes the challenge of what factors to consider and what metrics to optimize.

We investigate the Test Case Prioritization (TCP) problem in an industrial context characterized by frequent changes: Given a test suite, the problem is to find the best permutation of test cases according to a merit function f. The function f can depend on many factors, e.g. code coverage, test case diversity, if the test case has recently been run. The problem of TCP is not new, however its importance has increased in an industrial environment characterized by an iterative and incremental development process. In this rapidly changing development environment, multiple objectives are at play and it is not certain which ones are the most valuable. Our **research goal** is therefore to explore how different factors are affected by different prioritizations and if one mathematical approach can capture many factors.

Industrial Context: Westermo Network Technologies AB (Westermo) specializes in the development of robust industrial data communication products for industrial domains, such as on-board and track-side rail, energy distribution, etc. The SW in these products is the Westermo Operating System (WeOS), an embedded OS based on GNU/Linux developed by several software teams. WeOS is tested with a test automation framework developed in-house at Westermo. Testing is conducted using more than 20 test systems built up of network topologies with physical devices, see Figure 1. A physical test system has between 6 and 20 WeOS devices controlled by a PC running the test framework, which communicates with the devices over a serial console connection. Many test systems also have devices used to simulate loss of connection, traffic generators, as well as I/O for powering units on and off.

The rest of the paper is organized as follows. Sec. 2 presents related work. Sec. 3 introduces metrics and an optimization model. The design of the case study is in Sec. 4, with data collection in Sec. 5. Sec. 6 summarizes the results of our analysis. The contributions are discussed in Sec. 7 and Sec. 8 concludes the paper.

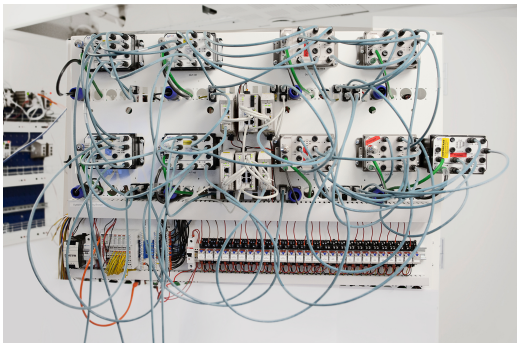


Figure 1: A Westermo test system with a network topology built up of switches, routers and other peripheral equipment.

2 RELATED WORK

In previous work, many techniques for solving TCP are proposed; Khatibsyarbini et al. [17] mention neural networks, greedy heuristics, genetic algorithms, etc. Yoo and Harman [38] mention weighted sums, Integer Linear Programming (ILP), meta-heuristics, multi-objective mathematical optimization, etc. A simple prioritization is the (additional) greedy approach, where the first test case is the one with the best value, the second has the second best value and so forth. If the value of the test cases is their coverage, then using the standard greedy approach sorts the test cases by how much each test case covers. This leads to test cases with small coverage being placed late in the suite. Greedy prioritizations are very fast and often very good [38].

Another relevant metric is the diversity of test cases. Two test cases are diverse if the distance between them, with respect to some metric, is large. Diversity metrics may involve coverage, and they may sometimes correlate [24]. Haghighatkhah [12] found that TCP based on diversity works well with a greedy approach. Thomas et al. [36] use topic models of linguistic data to prioritize test cases. This stems from the idea that similarly named test cases ought to test the same thing. Their technique outperforms standard techniques and has comparative results with history-based approaches.

Test cases need to be prioritized with respect to some metric, e.g., coverage, human or business importance [14], time since evaluation, diversity, interaction testing [38]. The tool currently in use at Westermo was implemented and evaluated about seven years ago [34]. The evaluation found that fault-detecting test cases were located early in the suite, that no test case had systematically been suppressed and that the need for manual work had been reduced.

If the goal of the test suite is to detect failing tests, then a proxy metric is needed because the true number of found faults would only be known if all test cases would be executed each night. Sometimes the focus instead lie on prioritizing test cases that have not been run for a long time – such a prioritization could target the time-since-run metric. Various methods for evaluating a prioritized test suite exist. In their survey, Heleno et al. [5] identify: Average Percentage of Faults Detected (APFD), the cost-cognizant weighted APFD ($APFD_C$), relative position, Average Percentage of Statement Coverage (APSC). Additionally, Khatibsyarbin et al. [17] also discuss coverage effectiveness and execution time. The most common evaluation techniques seem to be APFD and $APFD_C$ [5, 17]. The APFD metric can be seen as the area under the curve of a test case vs. the number of faults detected graph [27].

Metrics for coverage can also be constructed. A special version is when statements are used and then APSC is achieved. This reasoning can be used to construct any APxx metric that would be of interest. Based on the APFD, $APFD_C$ takes into account the cost and importance of test cases [7]. This prioritizes test cases deemed severe and de-prioritizes expensive test cases. When both costs and fault severities are identical, the formula reduces to the formula for APFD. Elbaum et al. [7] used a greedy heuristics based on $\frac{\text{severity}}{\text{cost}}$ to achieve good results. Hemmati [16] maintains that one may want to maximize the average distance between test cases given a diversity metric such as the Jaccard distance. Azizi and Do [2] claim that the most common similarity metric for TCP is the Jaccard distance.

Integer programming for regression testing is explored in previous work (e.g. by Mirarab et al. [23], Zhang et al. [39], Hao et al. [13], as well as Fu et al. [11]), and covers constraints such as limited time, limits in number of test cases to select, etc. As far as we can tell, these papers do not consider continuous integration (CI), or multiple heterogeneous test systems.

In CI, SW updates are frequent and developers want rapid feedback. We were unable to find previous work involving mathematical optimization methods for RT and CI, whereas Elbaum et al. [8] and Spieker et al. [30] have explored other approaches for RT and CI. A recent survey by Lima and Vergilio [20] states that the area of RT in CI is very new with most studies published after 2016. Most of the studies use a history-based approach. The most common evaluation metrics are time for executing the prioritization method or the test suite, and APFD or $APFD_C$.

Haghighatkhah [12] found that using historical data is very effective. When no previous data exists or is limited, one ought to strive for diversity. Later, one should incorporate test results data, while still aiming for diversity such that multiple test cases do not find the same fault. Liang et al. [19] use a lightweight technique that updates the prioritization whenever SW changes are submitted. Busjaeger and Xie [3] use machine learning while looking at code coverage, test script file path similarity, text content similarity, failure history, and test case age with promising results. Qu et al. [26], studied what happens when the execution of a test suite is divided over multiple machines. From a given prioritization, they split the execution of the suite over multiple test systems. Given the heterogeneity of test systems in our context this is not directly applicable for us.

3 METRICS AND OPTIMIZATION MODEL

In order to get an idea of the relevance of metrics used for both prioritization and evaluation, a survey with eleven open questions was sent to WeOS developers and testers, asking for factors of importance for regression test prioritization as well as ranking some pre-defined factors. The survey was completed by six respondents, which was enough to give us an indication that our metrics for prioritization and evaluation are relevant. For regression test prioritization, the respondents gave different factors of importance, such as taking into account previous failures, testing for changes in

WeOS code as well as in the test framework, testing for basic functionality, for test cases to run at least once a week, to test multiple subsystems, and for flaky test cases to be prioritised. For ranking of pre-defined factors, all respondents agreed that prioritising failing test cases, prioritising test cases that have not been run for several nights, large coverage of functional areas and prioritising test cases that target changed WeOS code, is at least 'valuable'.

The metrics in this paper are either *intersystem* or *intrasystem*, i.e., some look at the whole meta-suite and some instead give a value for each test system (that is then averaged). E.g., if one night we find 0 out of 1 fail on test system A and 19 out of 19 fails on test system B then we have found 50% of possible fails per test system and 95% of possible fails in total. Our proposed model aggregates most metrics into one, such that we may capture as many factors as possible.

3.1 Metrics

This subsection introduces the metrics used for prioritization.

Fail Count and Percentage: This can be counted as the percentage of fails detected of all possible. We want to maximize the average number of fails found per system (#f), average percent of fails found per system (%fs) and average percent of fails found in total (%ft).

APFD: An intrasystem metric that measures how early fault revealing test cases are placed. Test cases are sorted by fail-rate.

$APFD_C$: An intrasystem metric that measures how early in a suite fault revealing test cases are placed. This metric also involves the duration of test execution as a cost and prioritizes rapid test cases before slow ones. The implementation is inspired by Khowala [18].

APTU: To address the issue of fault detection over parallel heterogeneous test systems, we propose a metric APTU (Average Percentage of Time Used), which cares about the start time of a failing test case, regardless of the test system.

Age: We use two metrics for age. First, the mean number of days since the execution of all the test cases in the suite per system (#dt), a metric where low values are good. Second, we use the mean number of days since previous execution of the tests that were executed, (#de), a metric where high values are good in the sense that we prioritized old test cases.

Coverage: This metric measures how many of the possible functional areas of the SW that are covered by the test cases (cov). We report the average coverage per test system. At the company, a functional area could be a feature such as firewall or backup, and test cases are annotated with the areas they target.

Diversity: Using the mean Jaccard distance[1] between all run test cases, we get an intersystem metric. The diversity (div) metric uses the system, branch, and areas as elements that can differ.

3.2 Mathematical Model

We propose a mathematical optimization model for the TCP. The model gives each test case multiple costs and we want to minimize the sum of the costs. This means that a test case with a large cost is deemed important and placed early on in the test suite.

Intersystem cost: First, a cost c_i for each test case i, regardless of system, to capture the idea that test cases will hopefully be run on at least one system that night. This cost is multiplied by variable z_i, the earliest end time for the test case on any system.

$$c_i = (\,(\text{days\_last\_run}) \,/\, (\text{days\_last\_fail})\,) \times (\text{mods} + 1) \tag{1}$$

Here mods is the number of modifications associated with all areas covered by that test case since the latest execution.

Intrasystem cost: Second, a cost c_{ij} for running each test case i on each system j, based on its fail-rate f_{ij} and estimated execution time t_{ij}. To ensure that a test case that passes its first run is scheduled again, we impose a minimum fail-rate of 0.1% for all test cases. The cost c_{ij} is multiplied by variable x_{ij}, the test case's predicted end time, hence taking the sequence of the test cases on each system into account.

$$c_{ij} = \max(f_{ij}, 0.001) \,/\, t_{ij} \tag{2}$$

Penalty cost: Third, each test case i is given a cost for not being run on system j with time budget T_j. The cost d_{ij} is multiplied by variable u_{ij}, a binary variable that indicates whether a test case is expected to be run on a system (0) or not (1).

$$d_{ij} = ((\text{days\_last\_run})_j \,/\, (\text{days\_last\_fail})_j) \times (\text{mods} + 1) \times T_j \tag{3}$$

Variables

$x_{ij} =$ completion time of test case i on system j

$$y_{ijk} = \begin{cases} 1 & \text{if test case } i \text{ precedes test case } k \text{ on system } j \\ 0 & \text{otherwise} \end{cases}$$

$z_i =$ earliest completion time for test case i on any system

$$u_{ij} = \begin{cases} 1 & \text{if test case } i \text{ is not expected to run on system } j \\ 0 & \text{otherwise} \end{cases}$$

Parameters

$f_{ij} =$ fail-rate of test case i on system j

$t_{ij} =$ execution time for test case i on system j

$T_j =$ time budget for system j

$c_i =$ cost for running test case i on any system

$c_{ij} =$ cost for running test case i on system j

$d_{ij} =$ penalty cost for not running test case i on system j

$M =$ Big M, a sufficiently large number

Proposed model

$$\min\ h = \sum_i c_i z_i + \sum_{i,j} c_{ij} x_{ij} + d_{ij} u_{ij} \tag{4a}$$

$$\text{s.t. } x_{ij} + t_{kj} \le x_{kj} + \mathrm{M}(1 - y_{ijk}) \quad \forall\, i, j, i < k \tag{4b}$$

$$x_{kj} + t_{ij} \le x_{ij} + \mathrm{M} y_{ijk} \quad \forall\, i, j, i < k \tag{4c}$$

$$x_{ij} \le T_j + \mathrm{M} u_{ij} \quad \forall\, i, j \tag{4d}$$

$$z_i = \min_j \{x_{ij}\} \quad \forall\, i \tag{4e}$$

$$x_{ij} \ge t_{ij} \quad \forall\, i, j \tag{4f}$$

$$z_i \ge 0 \quad \forall\, i \tag{4g}$$

$$y_{ijk} \in \{0, 1\} \quad \forall\, i, j, k \tag{4h}$$

$$u_{ij} \in \{0, 1\} \quad \forall\, i, j \tag{4i}$$

(Note that constraint (4e) can be reformulated as a linear constraint.) The proposed model is classified as an ILP and the objective function (4a) factors in multiple costs and penalises test cases that are

[1]Other diversity metrics, e.g. Test Set Diameter [10], Cohen and Vitányi's [4] normalized compression distance, were not explored due to their computational complexity.

not run. Constraints (4b)–(4c) schedules each test case on each system, and constraint (4d) detects if a test case is unable to fit into the time budget. Constraint (4e) identifies the earliest completion time for each test case over all systems, and constraints (4f)–(4i) define all variable restrictions.

The proposed model gives raise to a large number of variables and constraints. Big M constraints are known to yield poor LP-relaxations and make the model hard to solve, and we therefore use a meta-heuristic to solve it.

We use the value of the objective function (4a) as an additional metric to compare prioritizations with. We call this metric *h-met*, our heuristic metric. The problem of finding a valid test suite is simple; just schedule the test cases in any order. Further, we see that from a given suite it is trivial to get to a new suite; just swap the order of two test cases on the same system. These insights lead us to try simulated annealing (SA) [35]. Perejo et al. [25] discuss alternative meta-heuristics to SA such as hill-climbing, tabu search, variable neighborhood search, evolutionary algorithms, and many more. Mansour et al. [21] tested several different test case selection algorithms including SA. Because of their objective function, they gain high coverage with a low number of test cases, that is, they maximized for coverage. They recommend SA over other algorithms when the test cases have different costs.

Phrased as a search-based software engineering problem [15], TCP could be explained as: (i) We *represent* the test suite as a 2D-array. The elements are test case tuples, each column represents a system, and each row represents the order of test cases for. The tuples contain costs for the test case, estimated runtime, test case number, and associated system. (ii) The *fitness function* used is (4a). From the array, we have the order of the test cases, runtimes, and costs, so it is easy to compute. Finally, (iii) the *manipulation operator* is to change place of the elements in the same row. Readers are reminded that, for SA, we sometimes accept changes that do not lead to improvements in the fitness function. As the iterations progress, the likelihood of accepting poorer suites decrease. This likelihood is modeled with "temperature" in a probability function.

We consider several SA configurations. First a random approach, SA-r. With a random start suite, its neighborhood consists of all test suites where all test cases are in the same order except two in the same system, that is we move to a neighbour by swapping two test cases in the same system. SA-r uses starting temperature 10^6 and end temperature 10^{-9}. The second approach, SA-c, warm starts with the cost prioritization since it approximates the system-wide cost c_{ij} well. This in an attempt to improve a not terrible starting solution quite fast. For SA-c, we use a smaller search space, only permutations with adjacent tests are considered. SA-c uses starting temperature 10^4 and end temperature 10^{-8}. We evaluate both approaches with 10^4 and 10^5 iterations, called short and long. We thus have four different SA prioritizations SA-rs, SA-rl, SA-cs, and SA-cl. An exponential temperature update is used due to the difference in magnitude between the starting and ending temperatures. The starting and end temperatures were discovered through empirical testing. More precisely we tested SA-rs and SA-cs on the training set with different starting and ending temperatures of $10^{-10}, 10^{-9}, \ldots, 10^9, 10^{10}$.

All SA approaches use $exp(-(\text{diff} / \text{temp}))$ as probability functions. Here diff is the difference in value of the objective function (4a), and temp is the current value of the temperature. This function is quite common for SA approaches as it ranges from 0 to 1 for positive differences between the current solution and the new solution. That is, we always accept a better solution than our current and reject worse solutions often but not always. More iterations lead to a higher temperature and thus lower and lower probability. The probability for rejection also increases if solutions worsen.

4 CASE STUDY DESIGN

Here we describe our case study design, covering elements inspired by case study guidelines from Runeson et al. [28].

We investigate TCP at Westermo as the contemporary software engineering phenomenon in the real-life setting. Therefore the case of the case study is the industrial TCP process at Westermo. The unit of analysis is the test results data from a period of 500 nights when no prioritization was needed (all test cases ran each night).

The TCP problem at Westermo is selected because of existing relationships among the authors. The first author is a thesis student, investigating the TCP problem given by the company supervisor (second author), the third author is acting as the academic supervisor for the student and the fourth author is a subject expert with experience of joint projects with the company.

Three recent surveys on TCP and RT are from Ali et al. [1], Lima and Vergilio [20], and Minhas et al. [22]. All of them agree on the need of further research on the topic in terms of understanding industrial factors of importance, such as test case volatility, feature coverage and other similar contextual factors. In their review paper covering regression testing, Yoo and Harman [38] found that there are few available data sets and more industry-academia collaboration needs to happen.

This work aims to answer two research questions. **RQ1**: How could one formulate a mathematical optimization model for Westermo's regression testing? **RQ2**: How are different metrics such as "time since latest run," coverage or fail detection rate affected by different prioritization approaches?

Westermo provided test data from a period of 500 days when no prioritization was needed as all test cases could be run each night [32]. There are about 5000 nightly suites in the data set, where a suite is a series of test cases on the same test system and the same branch. In the data set, there is one stable branch and one development branch. There are also dates for when the development branch was changed into the stable branch, that is to say when there was a new release. The two active branches in the data set is in sharp contrast to the almost 60 branches in use today [31]. Note that, since our work prioritizes over SW branches and test systems, only one test suite is generated per night. This can be seen as a "meta-suite" containing multiple "system suites".

For each test case executed, the following information is saved: which parameters were used, execution time, and the result of the test case (pass/fail/other). The data also contains information on when the SW and testware were changed. The following factors for prioritization can thus be used: estimated time for a test case, expected failure rate of a test case, how do changes between SW and testware interact, and the coverage and diversity of test cases.

A non-disclosure agreement was signed by the researchers outside the case company. The data was provided after evaluation that

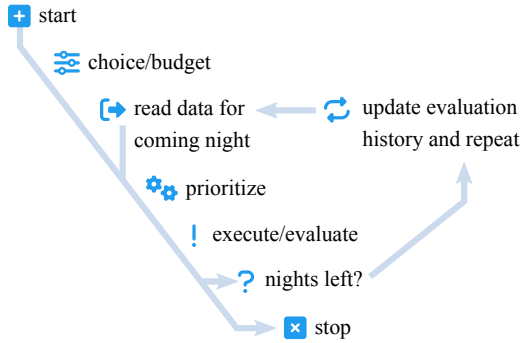


Figure 2: Flow chart for the implementation.

it does not harm anyone involved. This study has scientific value for the company and research on the topic in general, thus the study is considered beneficial.

5 DATA COLLECTION

We implemented a proof-of-concept prioritization tool in approximately 2000 lines of Python code, see overview in Figure 2 [9]. Initially, the commercial solver Gurobi with the Python API was explored. First, for a given night, the tool reads the data with the source code modification history of the previous day as well as the test case execution results from all test systems of the corresponding night. A prioritization is then created. Test cases that are viable for being in the prioritization are those that would have run that night or that have unused results from previous nightly testing. A test case's result is considered unused if the test case has not been run or updated in the previous night.

After prioritization, test cases are executed. For the 500 nights in the data, Westermo scheduled all test cases. For the simulation we used a budget of 50000 seconds, almost 14 hours, for each test system. This is enough for 95% of the nights and excludes some extreme values that would skew the time budget. The budget is representative of the testing starting at 6 PM and being done at 8 AM. The results were gathered using different prioritizations and different time budgets. The budgets used were 5%, 25%, 50% and, 100% of the base-line. The test cases are run test system-wise until that test system's time budget is spent. The test cases that are run on each test system had their execution history updated and their result for that particular test system is considered used.

In the evaluation stage, values of each metric are calculated and saved. The night is now over and the process is repeated.

In order to know which test cases that are candidates for being in nightly testing, the prioritization tool reads the data, e.g. when a new test case is introduced the tool would know about it. However, the results are only used in the evaluation stage and never in the prioritization stage with the exception of the ideal and worst possible prioritizations (described below). Also, the tool only uses test cases that were executed successfully (i.e., pass or fail). This is relevant since Westermo also provided data on other verdicts, e.g. when test cases could not run to completion.

The data set includes one development branch and one stable branch, and both branches receive code changes. In addition, the stable branch is sometimes updated with the SW from the development branch. This is to say that the SW had a new version released. When this happens, we also transfer the execution history from the development branch's test cases to the stable branch. Since each test system has unique hardware, their execution history is handled separately. This also helps with the fact that the number of test systems increase. In short, for each test case we have two branches and multiple systems to deal with, but not all test cases can run on all test systems or all branches.

At Westermo, the test cases are implemented as scripts, and some allow different parameter settings. Each script and parameter combination creates a new test case. So, for a script A and parameters $P1$ and $P2$, we have two test cases, $A - P1$ and $A - P2$.

The data used for prioritization is: (i) the historic mean fail-rate with respect to each test system, (ii) the historic mean execution time of the test case from the execution history with regard to each test system, (iii) number of modifications to areas associated with the test case since the last run regardless of the test system, (iv) days since the last run regardless of test system, and with regards to test system, and (v) number of days since the last fail regardless of test system, and with regards to test system. The prioritizations that rely on randomness (rdm and the four SA versions) were run five times, and the average is used as a comparison.

In addition to the four proposed SA prioritizations in the end of Section 3, we also explored nine reference prioritizations. These thirteen approaches represent rows in Table 1.

Ideal: We define the ideal prioritization as, test system-wise, sorted by if the test case will find a fail and then also by its duration. In this case, all fails are found early and efficiently. This prioritization is unrealistic in the sense that it will "cheat" and look into the future, so we use it as a reference only.

Worst: Similar to the ideal approach, this acts as a reference approach by placing failing tests at the end of the suites.

Random (rdm): Random is a classic reference approach [38].

Fail-rate (fr): Given the execution history of the test cases, we calculate the fail-rate of the test cases. We implement this prioritization as it is quite intuitive that a test case that usually fails might fail again the coming night.

Age: To avoid a test case not being run, the age prioritization sorts by descending number of days since latest execution.

Recent Failures (rec.f.): In this prioritization, the test cases are sorted by the amount of days since the latest failure. For realism, this prioritization was designed to be unaware of failures of test cases not executed in the simulations.

Modification (mod): The data contains information on SW modifications broken down by functional area. Recent and frequent changes are deemed more important, and the test cases are prioritized based on amount of modifications since latest execution.

Cost: Inspired by Elbaum et al. [7], we use a cost per test case: $\frac{\text{fail-rate}}{\text{execution time}}$.

Coverage (cov): Using the area words we can use an additive greedy heuristic which aims to maximize coverage. In a greedy way we add test cases based on additional coverage until everything is covered, at which point we reset the coverage and start over. As tie-breaker the fail-rate is used.

Table 1: Average metric value over all nights for all combinations of metric and prioritization with 5% of the time budget. Columns represent metrics (see Section 3.1). Rows represent prioritization approaches (see Sections 3.2 and 5). Best value per column (metric) is highlighted in bold, e.g., for the heuristic metric h-met the SA-rs was best. h-met is scaled by $\times 10^7$. High values are good for all metrics (+) except for #dt and h-met (-).

	#f (+)	%fs (+)	%ft (+)	APFD (+)	$APFD_C$ (+)	APTU (+)	#de (+)	#dt (-)	cov (+)	div (+)	h-met (-)
ideal	13.1	0.805	0.663	0.962	0.963	0.968	9.55	40.6	0.291	0.907	74.9
worst	0.01	0.004	> 0.001	0.071	0.060	0.457	10.5	83.5	0.0561	0.909	264
rdm	2.29	0.069	0.0571	0.503	0.503	0.761	18.9	18.8	0.226	0.908	16.7
f.r.	3.47	0.097	0.0805	0.590	0.587	0.771	6.94	62.1	0.150	0.909	220
age	2.30	0.088	0.0606	0.522	0.517	0.758	**21.8**	**11.1**	0.166	0.907	31.0
rec.f	3.15	0.086	0.0770	0.512	0.503	0.791	6.28	71.1	0.139	**0.909**	62.2
mod	3.13	0.118	0.0796	**0.609**	0.596	0.770	17.8	28.8	0.146	0.907	8.52
cost	3.56	0.095	0.0820	0.458	0.546	0.793	5.47	59.9	0.276	0.908	165
cov	0.84	0.031	0.0270	0.450	0.474	0.731	7.20	70.6	**0.318**	0.908	184
SA-rs	3.23	0.119	0.0877	0.563	0.619	0.818	16.5	20.2	0.240	0.907	**6.68**
SA-rl	**3.93**	**0.137**	**0.110**	0.561	**0.631**	**0.825**	16.0	22.9	0.219	0.907	6.75
SA-cs	3.49	0.093	0.0806	0.461	0.550	0.793	5.39	60.0	0.275	0.908	167
SA-cl	3.60	0.096	0.0825	0.457	0.548	0.793	5.50	59.4	0.285	0.908	157

6 DATA ANALYSIS & RESULTS

This section presents and compares the results from the metrics and prioritizations. We also cover the durations needed to build suites, and compare the performance of the prioritizations.

Table 1 presents the average performance over all nights, for each combination of metric and prioritization when a limited time budget equal to 5% of the available time was used. The best value is highlighted with bold text; ideal and worst are not considered, they are only used as references.

To our surprise, the h-met value for cost prioritization is lower than SA-cs even though SA-cs starts from the cost prioritization and seeks a lower h-met value. A possible explanation is the non-determinism of SA, and that these prioritizations handle history differently. This is most prominent with a low time budget. We also see that SA-rl is the best in many metrics. For other time budgets (25, 50 and 100%), SA-rl also fared well, in particular for fault finding metrics, and $APFD_C$. Across time budgets, the recent fail and modification prioritization approaches were the best in some metrics (some of the APxx metrics as well as #de and div). Further, the modification prioritization gave the best h-met for the 50% budget. We observed that h-met seems to decrease as the time budget increases, which is better due to the fact that more test cases are expected to run each night and thus fewer test cases to be penalized. We also note that modification prioritization achieves good marks in the h-met metric. As could be expected, the coverage prioritization was always the best for maximizing the coverage metric. Similarly, the age prioritization always gave the lowest mean number of days since executing for the entire suite. Random prioritization has the second lowest age of executed test cases with SA coming after. The diversity metric decreases slightly with increased time budget likely due to the test cases being somewhat similar. The difference in diversity between prioritizations is small and we are tempted to suggest avoiding this metric. Interestingly, the poorest prioritization was achieved by aiming at diversity followed by recent fails. We can see that ideal is the prioritization that has the best coverage. In contrast, coverage prioritization is bad at finding fails.

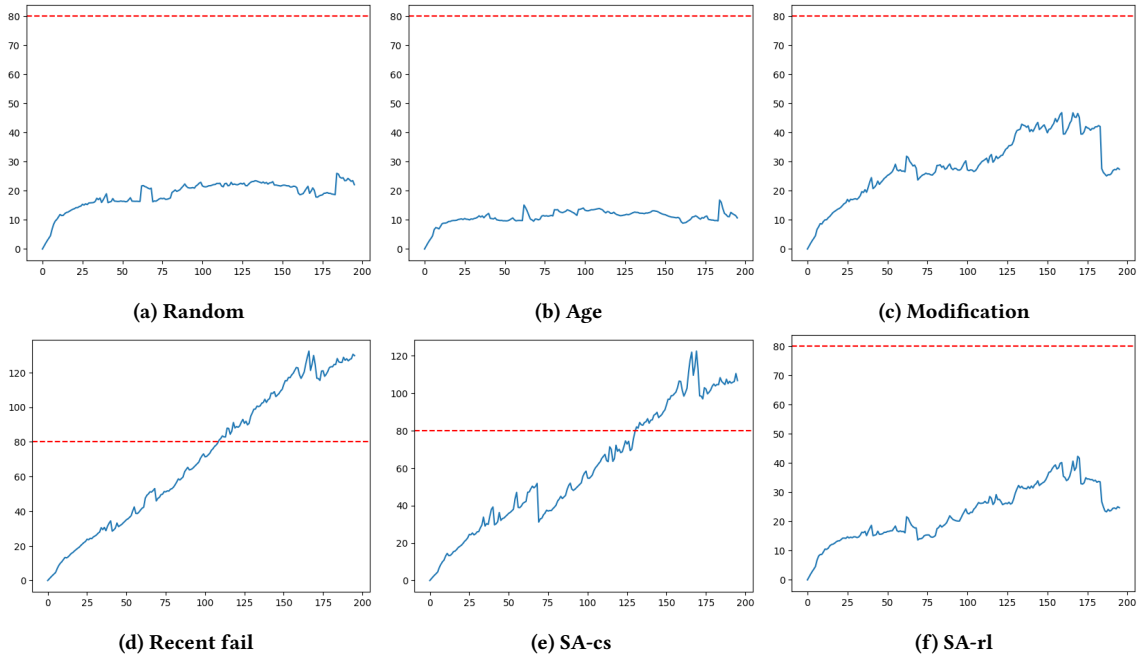
Figure 3 illustrates the performance of six prioritization approaches with respect to the metric of number of days since last execution. As could be expected, a prioritization that only focuses on age results in testing with a low average age (see Fig. 3b), so with only that goal in mind, that prioritization is meaningful. Also, the figure clearly illustrates how well random and SA-rl perform over time with respect to this metric. However, the recent fail and SA-cs prioritizations rapidly degrade and surpasses 100 days in the end of the curve. Hence, they are about five times as poor as random with respect to this metric. As argued by Harman [14], single objective prioritization is unlikely to be practical. Figure 3 gives a mixed message; if the objective function and metric go hand in hand, the outcome could be desirable. However, it is not certain that all multi objective prioritizations result in desirable outcomes.

The duration of the suite generation (after data had been collected) was very low (from 0.8 to 4.3 ms) for the simplest prioritizers (ideal, worst, rdm, fr, age, rec.f, mod, and cost), 1.09 seconds for cov, about 15 seconds for the short SA approaches, and almost 100 seconds for the long SA approaches. These are low values when compared to 14 hours of nightly testing.

We compared the prioritizations using performance profiles [6] and found that the ideal approach was best in about half of the metrics, the modification prioritization was the approach that was "least worst" when compared to the others, and age prioritization was quite good. Ideal prioritization quickly found fails, but did not take into account all factors of TCP to be considered the best. Cost prioritization was slightly better than random prioritization. SA approaches were not in the top. However, if we removed time for generating a suite in the performance profile analysis, then SA-rl and SA-rs both rose to the top. SA-c approaches performed poorly. Finally, the worst prioritization was indeed the worst.

7 DISCUSSION

The random prioritization was bad at finding fails. However, it resulted in good coverage and did not let any test case get too old. Modification prioritization successfully found faults since changes



(a) Random (b) Age (c) Modification

(d) Recent fail (e) SA-cs (f) SA-rl

Figure 3: Average number of days since last execution of the test cases in the suite (y-axis), with a 5% time budget, over time (x-axis), when using six prioritization approaches. Please note that the figures have different scales on the y-axes – an average age of 80 days since last execution is indicated with dotted lines.

increase the chance for new bugs. Further, all areas are updated at some point, so no test case was completely forgotten. For finding fails, this prioritization is above average, and for some time budgets best. Age prioritization did not forget any test cases, thereby also yielding good coverage. However, it was poor at prioritizing failing tests. As expected, it prioritized older test cases early and thus have great values for the age metrics. Using the functional areas of the SW as a basis for a greedy additional coverage prioritization was not very successful, except for maximizing coverage.

SA-r performed well, in part because the prioritization with the lowest h-met, modification prioritization, performed well. It also seems as the other parts of the model correlated with prioritizations age, fail-rate, and cost, worked in great unison to make SA-r the best performing prioritization, excluding duration. Compared to SA-r, SA-c did not perform well. We speculate that the neighborhood was too small – i.e. it was hard for SA-c to change the suite since each iteration only involved changing two adjacent test cases.

To evaluate diversity, the mean Jaccard distance between the test cases was used, which was not very valuable. If other features in a distance metric were used, then perhaps the results would have been better. It is also possible that the use of Westermo's areas is a too coarse basis for diversity. A possible approach for a good diversity metric is to use the test set diameter [10]. However, this metric is computationally intensive.

A weakness of the proposed optimization model is that coverage is not accounted for. However, the results from the SA approaches show pretty good functional coverage metrics when compared to the coverage-only approach. The mathematical model was however discarded due to it being computationally intractable, even without considering coverage. SA is not limited by the preference of a linear objective function, and future work could explore the possibility to incorporate more factors into the objective function.

The lack of randomness in many of the prioritizations could be seen as a shortcoming. Coverage prioritization seems to suffer from this as the greedy additive method only looks at the test areas during a single prioritization and not on coverage over many nights. It is also possible that the fail-rate, recent fail, and cost prioritizations would benefit from it – otherwise slow test cases might not be re-selected if they pass the first time they run.

For future work, it would be interesting to make a deeper investigation in the manipulation operator, i.e. the neighborhood of a suite. The SA-c approach could be evaluated with a larger neighborhood and SA-r with a smaller. This could lead to a better understanding of the value of a warm start, and the best manipulation strategy.

From a human factors perspective, it would be relevant to know how Westermo staff would feel about using SA in their daily work. E.g. would they trust the level of randomness involved? Would they see it as fair?

With SA-rs taking about 15 seconds to prioritize one meta-suite, it would be possible to have a human in the loop. The human could look at the meta-suite, change some parameters, and iteratively co-create a meta-suite with a tool. This approach could lead to biased suites, but perhaps improved human satisfaction.

This study presents several implications for industrial practitioners, many of these repeat or confirm findings from previous research. First, formal optimization for test case selection is only suitable when there are few test cases, code branches and test systems, due to the duration required for solving to optimality. Second,

the heuristic method of using simulated annealing with a multi-objective (or weighted objective) function is a feasible and rather rapid approach for prioritizing test cases (given that historic data collection is fast, which it may or may not be in practice). Third, practitioners ought to be careful what they wish for. As Norbert Wiener (possibly the first to consider ethical implications of automation) warned, introducing automation is like getting wishes satisfied by dark magic [37]. In particular, the study at hand found that a test case prioritization focusing on only one factor (failing test cases) may come with unexpected penalties for others (average days since last execution), as illustrated in Fig. 3d. This challenge is well studied in other domains such as avionics [29] and ethical aspects of test case selection has also been partially covered [33]. In practice this means that practitioners ought to strive for a multi-objective goal function, and monitor any potential performance degradation aspects of the automation – in particular average days since last execution.

8 CONCLUSION

The research goal of this study was to explore how different factors (e.g. test coverage) are affected by different prioritizations, and if it would be possible to capture many factors with one prioritization. With respect to RQ1, we formulated a mathematical optimization model, equations 4a–4i, which captured many factors of TCP in the industry context at Westermo. However, the model did not scale well enough. Instead, solving the model heuristically using SA was shown to be a viable option for a practical application. Regarding RQ2, we could see that fail-rate, time since latest run, and coverage was highly dependent on the prioritization used. This is particularly important to consider when the budget is low. Our metric for diversity was not very successful. On this data set, we see that modification prioritization is the overall best choice. If the user can spare some time for planning, SA-r is the preferred choice.

ACKNOWLEDGMENTS

The work was sponsored by Westermo, the Swedish Knowledge Foundation through grant 20150277 (ITS ESS-H), and the AIDOaRt project, an ECSEL Joint Undertaking (JU) under grant agreement No. 101007350.

REFERENCES

[1] Nauman Bin Ali, Emelie Engström, Masoumeh Taromirad, Mohammad Reza Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. 2019. On the Search for Industry-Relevant Regression Testing Research. *Empirical Softw. Engg.* 24, 4 (2019), 2020–2055.
[2] Maral Azizi and Hyunsook Do. 2018. Graphite: A greedy graph-based technique for regression test case prioritization. In *ISSREW'18*. IEEE.
[3] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: an industrial case study. In *FSE'16*. ACM.
[4] Andrew R Cohen and Paul MB Vitányi. 2014. Normalized compression distance of multisets with applications. *IEEE PAMI* 37, 8 (2014).
[5] Heleno de S. Campos Junior, Marco Antônio P Araújo, José Maria N David, Regina Braga, Fernanda Campos, and Victor Ströele. 2017. Test case prioritization: a systematic review and mapping of the literature. In *SBES'17*.
[6] Elizabeth D Dolan and Jorge J Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical programming* 91, 2 (2002).
[7] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE'21*. IEEE.
[8] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *FSE'14*.
[9] Eric Felding. 2022. *Mathematical Optimization and the Test Case Prioritization Problem.* Master's thesis. Linköping University.
[10] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test set diameter: Quantifying the diversity of sets of test cases. In *ICST'16*. IEEE.
[11] Wenhao Fu, Huiqun Yu, Guisheng Fan, Xiang Ji, and Xin Pei. 2017. A regression test case prioritization algorithm based on program changes and method invocation relationship. In *APSEC'17*. IEEE.
[12] Alireza Haghighatkhah. 2020. *Test case prioritization using build history and test distances.* Ph. D. Dissertation. University of Oulu, Finland.
[13] Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, and Tao Xie. 2015. To be optimal or not in test-case prioritization. *IEEE Transactions on Software Engineering* 42, 5 (2015).
[14] Mark Harman. 2011. Making the case for MORTO: Multi objective regression test optimization. In *ICSTE'11*. IEEE.
[15] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001).
[16] Hadi Hemmati. 2019. Advances in techniques for test prioritization. In *Advances in Computers*. Vol. 112. Elsevier.
[17] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang NA Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (2018).
[18] Ketan Khowala. 2012. *Single Machine Scheduling: Comparison of MIP Formulations and Heuristics for Interfering Job Sets.* Arizona State University.
[19] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: continuous prioritization for continuous integration. In *ICSE'18*.
[20] Jackson A Prado Lima and Silvia R Vergilio. 2020. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Information and Software Technology* 121 (2020).
[21] Nashat Mansour, Rami Bahsoon, and Ghinwa Baradhi. 2001. Empirical comparison of regression test selection algorithms. *Elsevier JSS* 57, 1 (2001).
[22] Nasir Mehmood Minhas, Kai Petersen, Jürgen Börstler, and Krzysztof Wnuk. 2020. Regression testing for large-scale embedded software development – Exploring the state of practice. *Information and Software Technology* 120 (2020), 106254.
[23] Siavash Mirarab, Soroush Akhlaghi, and Ladan Tahvildari. 2011. Size-constrained regression test case selection using multicriteria optimization. *IEEE transactions on Software Engineering* 38, 4 (2011).
[24] Debajyoti Mondal, Hadi Hemmati, and Stephane Durocher. 2015. Exploring test suite diversification and code coverage in multi-objective test case selection. In *ICST'15*. IEEE.
[25] José Antonio Parejo, Antonio Ruiz-Cortés, Sebastián Lozano, and Pablo Fernandez. 2012. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing* 16, 3 (2012).
[26] Bo Qu, Changhai Nie, and Baowen Xu. 2008. Test case prioritization for multiple processing queues. In *ISISE'08*. IEEE.
[27] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, 10 (2001).
[28] Per Runeson, Martin Höst, Austen Rainer, and Bjorn Regnell. 2012. *Case Study Research in Software Engineering: Guidelines and Examples.* John Wiley & Sons.
[29] Nadine B Sarter, David D Woods, Charles E Billings, et al. 1997. Automation surprises. *Handbook of human factors and ergonomics* 2 (1997).
[30] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *ISSTA'17*.
[31] Per Erik Strandberg. 2021. *Automated System-Level Software Testing of Industrial Networked Embedded Systems.* Ph. D. Dissertation. Mälardalen University.
[32] Per Erik Strandberg. 2022. *The Westermo test results data set.* Technical Report. GitHub, https://github.com/westermo/test-results-dataset.
[33] Per Erik Strandberg, Mirgita Frasheri, and Eduard Paul Enoiu. 2021. Ethical AI-Powered Regression Test Selection. In *AITest'21*. IEEE.
[34] Per Erik Strandberg, Daniel Sundmark, Wasif Afzal, Thomas J Ostrand, and Elaine J Weyuker. 2016. Experience report: Automated system level regression test prioritization using multiple factors. In *ISSRE'16*. IEEE.
[35] El-Ghazali Talbi. 2009. *Metaheuristics : from design to implementation.* John Wiley & Sons.
[36] Stephen W Thomas, Hadi Hemmati, Ahmed E Hassan, and Dorothea Blostein. 2014. Static test case prioritization using topic models. *Empirical Software Engineering* 19, 1 (2014).
[37] Norbert Wiener. 1973. Cybernetics–2nd edition: Or the control and communication in the animal and the machine.
[38] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Wiley STVR* 22, 2 (2012).
[39] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. 2009. Time-aware test-case prioritization using integer linear programming. In *ISSTA'09*.